Vector search in Apache Cassandra
How LLMs + vector search make AI tractable for everyone
Old school AI
AI with LLMs
Cassandra's unified compaction strategy, introduced in version 4.0, aims to simplify the compaction process and improve overall database performance. It combines several compaction strategies into a single, more flexible approach. Here's a brief summary of Cassandra's unified compaction strategy:

1. **Background:**
   - In earlier versions of Cassandra, multiple compaction strategies existed, each with its own trade-offs and configuration parameters.
   - Unified compaction was introduced to streamline compaction and make it easier to manage.

2. **Single Compaction Strategy:**
   - Unified compaction merges multiple compaction strategies into a single, unified approach.
   - It combines Size-Tiered, Time Window, and Leveled compaction into a more versatile strategy.

3. **Dynamic Configuration:**
   - Unified compaction allows for dynamic configuration based on workload and performance requirements.
   - Users can adjust compaction parameters such as the min and max compaction throughput, compaction window size, and others to fine-tune performance.
Cassandra's Unified Compaction Strategy (UCS) is a new approach to compaction in Cassandra that combines aspects of both tiered and leveled compaction strategies. It introduces sharding, adaptability, and aims to optimize read and write performance based on user-defined parameters. Here's a summary of how UCS works:

1. **Generalization of Compaction**: UCS generalizes tiered and leveled compaction strategies, treating them as the same by observing that both create levels based on the size of sstables and trigger compactions when a certain number of sstables are present on one level.

2. **Density-Based Levels**: UCS categorizes sstables into levels based on the logarithm of sstable density, with a user-defined fanout factor ($f$) as the base of the logarithm. Each level triggers a compaction when it contains a specified number ($t$) of overlapping sstables. The values of $f$ and $t$ determine the strategy's behavior, allowing users to choose between leveled and tiered compaction.

3. **Sharding**: UCS splits sstables at specific shard boundaries based on the density of an sstable. These shard boundaries increase with the density of the sstable, enabling concurrent compactions.

4. **Size-Based Levels**: If we ignore density and splitting, UCS groups sstables into levels based on their size relative to a memtable flush size. This allows UCS to determine the level of an sstable based on its size compared to a predefined flush size.
AI with LLMs + context
Finding the right context
Embedding turns any text into vectors
Embeddings capture semantics

- “Man bites dog” vs “dog bites man”
- “Tourism numbers are collapsing” vs “Travel industry fears Covid-19 crisis will cause more companies to enter bankruptcy”
- “I need a new phone” vs “My old device is broken”
Vector similarity

\[
\text{vector 1} \cdot \text{vector 2} = F_{32}
\]
Vector search 101
KNN
**HNSW**

As seen in:

- Lucene
  - Elastic, Solo, OpenSearch
  - Mongo?
- Weaviate
- Qdrant
- Astra (June 2023)
- ...

[Diagram of HNSW layers with nodes and connections]
SAI (Storage Attached Indexes)
SSTable lifecycle
Global ANN everywhere

```sql
SELECT * FROM demo
ORDER BY embedding ANN OF ?
LIMIT 10
```

Composes with partitioning

```sql
SELECT * FROM demo
WHERE partition_id = ?
ORDER BY embedding ANN OF ?
LIMIT 100
```

Composes with other SAI indexes

```sql
SELECT * FROM demo
WHERE (c1 = ? AND c2 = ?)
  OR c3 = ?
ORDER BY embedding ANN OF ?
LIMIT 5
```
Executing ANN queries

1. Apply SAI predicates across all stables
2. Query each ANN index for top K relevant rowIDs (or brute force it)
3. Merge by score
4. Reorder by primary key
5. Send results to coordinator
HNSW in production
100M Wikipedia vectors
DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node

Suhas Jayaram, Subramanya
Carnegie Mellon University
suhas@cmu.edu

Devvrit*
University of Texas at Austin
devvrit.03@gmail.com

Rohan Kadekodi*
University of Texas at Austin
rak@cs.texas.edu

Ravishankar Krishnaswamy
Microsoft Research India
rakri@microsoft.com

Harsha Vardhan Simhadri
Microsoft Research India
harshaas@microsoft.com

Abstract

Current state-of-the-art approximate nearest neighbor search (ANNs) algorithms generate indices that must be stored in main memory for fast high-recall search. This makes them expensive and limits the size of the dataset. We present a new graph-based indexing and search system called DiskANN that can index, store, and search a billion point database on a single workstation with just 64GB RAM and an inexpensive solid-state drive (SSD). Contrary to current wisdom, we demonstrate that the SSD-based indices built by DiskANN can meet all three desiderata for large-scale ANNs: high-recall, low query latency and high density (points indexed per node). On the billion point SIFT1B bigann dataset, DiskANN serves > 5000 queries a second with < 3ms mean latency and 95%+ 1-recall@1 on a 16 core machine, where state-of-the-art billion-point ANNs algorithms with similar memory footprint like FAISS [18] and IVFOADC+G+P [8] plateau at around 50% 1-recall@1. Alternately, in the high recall regime, DiskANN can

SPANN: Highly-efficient Billion-scale Approximate Nearest Neighbor Search

Qi Chen1,*, Bing Zhao1,2,† Haidong Wang1 Mingjin Li1 Chuanjie Liu1,3,† Zengzhong Li1 Mao Yang1 Jingdong Wang1,4,*,†
1Microsoft 2Peking University 3Tencent 4Baidu
1cheqi, haidwa, mingqili, jasol, moyangj1@microsoft.com
2its.bingzhao@pku.edu.cn 3li.chuanjie@outlook.com 4wangjingdong@outlook.com

Abstract

The in-memory algorithms for approximate nearest neighbor search (ANNS) have achieved great success for fast high-recall search, but are extremely expensive when handling very large scale database. Thus, there is an increasing request for the hybrid ANNS solutions with small memory and inexpensive solid-state drive (SSD). In this paper, we present a simple but efficient memory-disk hybrid indexing and search system, named SPANN, that follows the inverted index methodology. It stores the centroid points of the posting lists in the memory and the large posting lists in the disk. We guarantee both disk-access efficiency (low latency) and high recall by effectively reducing the disk-access number and retrieving high-quality posting lists. In the index-building stage, we adopt a hierarchical balanced clustering algorithm to balance the length of posting lists and augment the posting list by adding the points in the closure of the corresponding clusters. In the search stage, we use a query-aware scheme to dynamically prune the access of unnecessary posting lists. Experiment results demonstrate that SPANN is 2× faster than the state-of-the-art ANNS solution DiskANN to reach the same recall quality 90%
DiskANN
Product quantization

Original vector: d-dimensional

Decompose into M d’-dimensional subcomponents

Replaced with id of nearest centroid
JVector

About

JVector: the most advanced embedded vector search engine

- java
- search-engine
- machine-learning
- ann
- knn
- similarity-search
- vector-search

Releases

- Create a new release
QPS on Deep100M dataset (24GB Macbook)

- Lucene: 21.7
- JVector: 275.5
Closing thoughts
The fine print
(things that don’t work yet)

• Paging
• CL > 1
• … That’s it
Lessons learned

• Not correct to apply indexes to SSTables separately
• OpenAI embeddings are ridiculously overparameterized
  • Use e5-v2
• Kmeans++ doesn’t need a lot of iterations
• Vamana needs room to add edges
• Surprisingly hard to predict how many nodes will be visited when searching for the top K of B items in a graph of N
Where’s the code?
- JVector: https://github.com/jbellis/jvector/
- Cassandra + vector search (today): https://github.com/datastax/cassandra/tree/vsearch
- Cassandra + vector search (RSN): https://issues.apache.org/jira/browse/CASSANDRA-18557